Received: 15 June 2018 | Revised: 20 September 2018 | Accepted: 30 October 2018
DOI: 10.1002/cpe.5307

SPECIAL ISSUE PAPER

File retrieval and storage in the open source cloud tool using digital bipartite and digit compact prefix indexing method

P. Priya Ponnuswamy[1] | R. Vidhya Priya[2] | C.P. Shabari Ram[1]

[1]Department of Computer Science and Engineering, PSG Institute of Technology and Applied Research, Coimbatore, India
[2]PSG College of Technology, Coimbatore, India

Correspondence
P. Priya Ponnuswamy, Department of Computer Science and Engineering, PSG Institute of Technology and Applied Research, Coimbatore-641 062, India.
Email: priyabaskii@gmail.com

Summary

Cloud computing plays a major role in data centers, and its important services are used by the data centers. Data centers are large number of computers that are interrelated to make the cloud services available. Data centers are increasing nowadays to provide a platform to store files, data applications, etc. Data centers with large storage capacity are the foundation for cloud computing services such as web searching and network storage. Data traffic takes place within and outside the data centers and needs to forward data between servers in the network. In this paper, a new method called digital bipartite and digit compact prefix with hashing technique is used to store and retrieve data in the cloud. The idea is to create a volume of storage for a user within the data center to store their files. Digital bipartite and Digit compact prefix are implemented to store and retrieve the user files efficiently. The implementation is done in OpenStack with a multinode setup. The proposed method compares the insertion time, search time, and throughput with the number of nodes, the number of compute instances, and the number of files for the two algorithms.

KEYWORDS

cloud, data center, hashing techniques, open-source cloud system, OpenStack

1 | INTRODUCTION

Nowadays, cloud computing plays a vital role in accessing hardware and software resources through the Internet. The cloud offers convenient, on-demand access to a set of resources such as networks, computers, storage, applications, etc, as a service over the Internet. To provide these services, data centers play an important role. Data centers consist of hundreds of servers interconnected and some petabytes of storage maintained in a large-scale heterogeneous distributed system. Efficiently managing these servers for storage is a primary challenging one. By storing their data in the cloud, the users can be comforted from data maintenance and storage. In distributed systems, a hash table with indexing is one method to efficiently manage the stored files in large-scale systems such as data centers. The important property of the hash table with indexing method is that it can store and retrieve user files easily by minimizing the table structure and avoiding collisions. In cloud data centers, data storage and retrieval can be the biggest task because all the cloud servers are not on the same network. Data retrieval in cloud data centers is to be considered so that the maximum number of users can retrieve and store files in an effective manner. To maintain data in the data center, the cloud systems depend on distributed file systems. Examples of such systems include Google File System, MapR File System, Ceph File System, cluster file systems, IBM General Parallel File System, Parallel Virtual File System, and Hadoop Distributed File System. There are many techniques and algorithms available for cloud information retrieval. One example is for a given condition, the equivalent information is obtained from the file system. Another is in open-source cloud systems such as Amazon's EC2, where many clients can upload their own application in similar cloud machines. Because different client information is present in different data centers, monitoring all the data or information in different data centers becomes more difficult. Hence, a more effective data access method is needed. A new method in the cloud system has been developed. In this method, data center nodes are considered compute nodes, and these compute nodes are organized in a hierarchical structured network. Each data center node creates its own index to speed up the time of insertion and search of the files. A centralized index is made by considering only a portion of its own index in the network. Each data center node maintains a part of the global index. This type of storage and retrieval is impossible in large cloud systems.[1,2]

2 | RELATED WORK

Many researchers introduced algorithms for search and retrieval of data in cloud data centers. Lu and Shen[3] introduced an algorithm called key negotiation. This algorithm works in the shared cloud storage system with distributed hash table (DHT)–based networks. In key negotiation, bloom filter and B+ search tree data structures are used to store and update the data. The data and the key are processed and uploaded in the DHT table. An authorized user can query to obtain the data; the lookup operation is done by combining the B+ tree and the bloom filter. To update their data, another method called rsync algorithm is used. In this method, the file uploading and downloading times are measured with respect to file size. An algorithm called fuzzy keyword search and gram-based fuzzy set construction is also introduced to search in the cloud environment.[4,5] In this method, the focus is on reducing the space of indexing in the storage. The index is created by computing the trapdoor with the key and storing it in the cloud server. When an authenticated user types a word, then the edit distance is calculated, and it is transformed into a fuzzy set. Then, the trapdoor is generated and submitted to the cloud system. Once the server receives a query, the server checks whether there is any match with the index and returns a result if a match is found. The bloom filter is also used with the fuzzy keyword set to search. Users can store their files in the cloud, and a group of users can retrieve their files using a symbol-based tree traverse search method. In this method, a multiway tree is constructed for storing the fuzzy keyword. Pagh and Rodler[6] proposed a method called cuckoo hashing in which two hash tables are created. During insertion, it checks the position in the first hash table, and if it is filled with data, then it checks in the second hash table; the search continues until it finds an empty space. Before storing data in the cloud storage, encryption is done by a latent semantic method, and the index is maintained for the unique words. When a user searches for a keyword, the token algorithm computes the hash value in the cloud server. Then, the algorithm computes the sequence of hashes in the hash table to retrieve the files. The retrieved file mapped with the identifier and retrieve the documents with the vector coordinates. In another proposed method,[7] the file is initially loaded on the server and is split into chunks, which are then organized into a tree structure. The hash algorithm applied to all the chunks, pairing the hash and concatenated to get a new hash. The new hash is again hashed until 512 bits is generated for the entire file. To retrieve a file, a file name is mapped with the file stream, and the file stream is downloaded.

Proposed Work

Cloud application supports deploying a storage system and requires high scalability and throughput. Cloud storage systems are introduced to satisfy the requirements of data-exhaustive applications such as managing high-end storage and easy availability with low latency. Cloud users can store their data on remote servers. Files saved in cloud can be easily accessed and available through a network. A data center is also designed to provide efficient platform for data storage. To store and retrieve the data in the data center, data forwarding is to be done in the data center. The layout of the data center is switch-centric with fat-tree topology. In cloud, compute nodes act as a resource pool, and the number of nodes can be increased or decreased based on the demand. In the existing solution, if the volume of files stored in a storage device is too high, efficient retrieval and storage of files become an issue.[8-11]

Figure 1 shows the instance and volume creation in different compute nodes. This paper introduces two algorithms for storing and retrieving in the cloud storage platform. The controller allocates instances to compute nodes. The process of finding the compute nodes and allocating the virtual machine (VM) is done by the controller node. The compute node act as a server and the controller node acts as the index server. The compute nodes and VMs are assigned a unique number to identify the user files effectively, as shown in Table 1. A large number of user files are stored in compute nodes, where the volume is created in a cloud storage system. Inserting and searching a file in the storage use a new method and an optimized one.

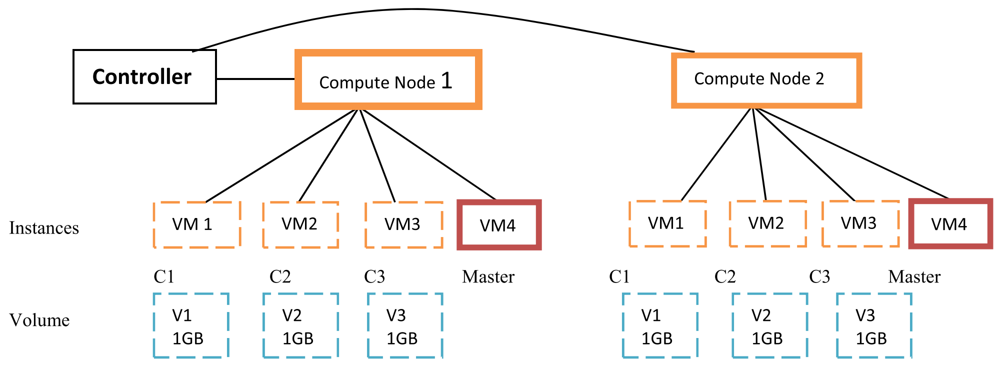

FIGURE 1 System structure of VM creation

TABLE 1 Unique ID for compute nodes and VMs

Compute 1: 00	Compute 2: 01	Compute 1: 10	Compute 2: 11
VM1 : 000	VM1 : 100	VM1 : 000	VM1 : 100
VM2 : 001	VM2 : 101	VM2 : 001	VM2 : 101
VM3 : 010	VM3 : 110	VM3 : 010	VM3 : 110
VM4 : 011	VM4 : 111	VM4 : 011	VM4 : 111

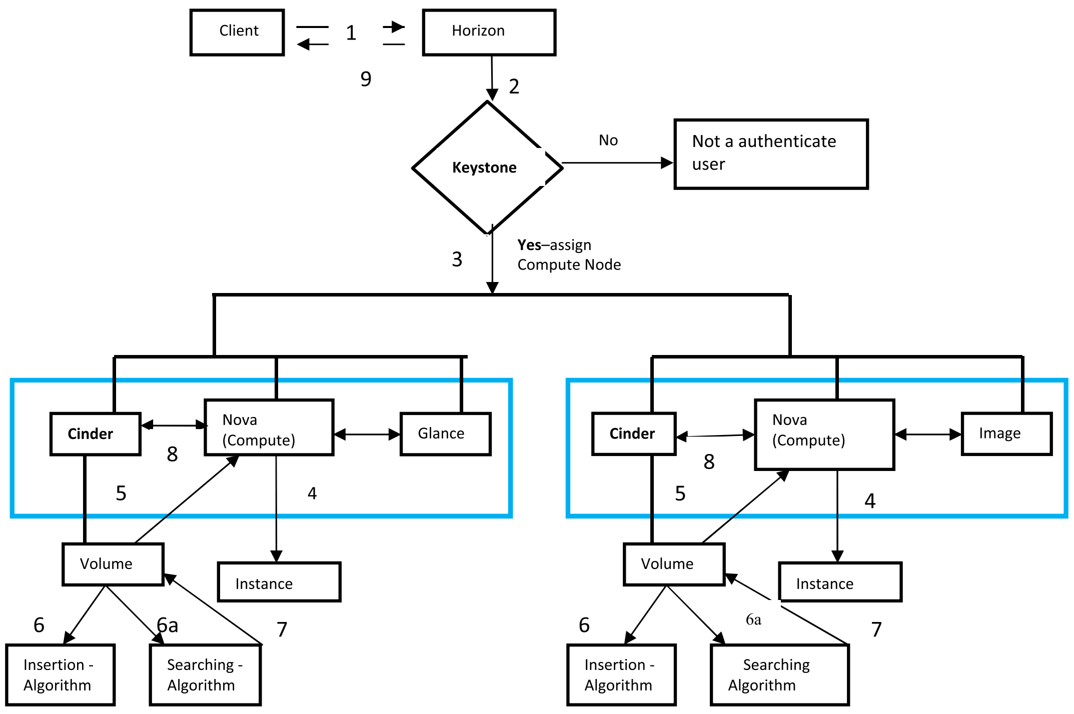

FIGURE 2 Framework of the proposed method

In cloud, the data center consists of one controller and four compute nodes. To search the files quickly, the local index must be maintained in all the compute nodes, and the global index must be maintained in the controller node. The proposed algorithm effectively works for storing and retrieving the large-scale files or applications in the cloud storage system.

The VM instances are created in each compute node, and they are called c1, c2, c3, and master. In each compute node, a master node will be maintained. The controller node acts as an entry point for the user and the administrator. Figure 2 shows the workflow of the proposed system, which is explained as follows. When a user is trying to access the cloud system, the horizon provides a web-based dashboard to use the services available in OpenStack. The keystone service in OpenStack decides and checks whether the user is authenticated or not with their username and password. If the user is authenticated, the user can be allowed access to the VM instances. The VM instances are allocated to any of the compute nodes. Through the allocated instances, the user can store, access, and retrieve their files and data. For each instance in a compute node, separate storage, ie, a volume of 1 GB, for each user is maintained by the service cinder. While storing and retrieving the user files, the digital bipartite and digit compact prefix indexing algorithm is to be executed in each compute node, and their meta data are maintained in the controller node. The metadata contain the ID generated by the compute node for VM, the compute node ID, and the file name and details about the file. When an authorized user submits a request to retrieve the stored files, the controller node transfers control to the particular compute node, and the master node sends a request to the particular node. The special search algorithm is maintained in all the master nodes to quickly retrieve the search results.

Whenever a VM is created, a unique ID will be generated and inserted automatically in the controller node. When a VM shuts down, the generated ID will be maintained in the controller node to find the user volume efficiently. Adding and deleting VMs concern only the IDs maintained; balancing of nodes would be done with minimum swapping.

The proposed method provides infrastructure as a service in the cloud. The user can get a VM to access, store, and retrieve their specific files that they are interested in.

3 | THE PROPOSED METHOD

3.1 | Digital bipartite insertion and searching

The digital bipartite search tree is similar to a binary search tree. It is used to organize the files in the cloud storage. The levels in the tree are organized based on the random numbers generated for a particular file name. For every level of the tree, a file-name random number is inserted by filling the left and the right child nodes. To build a tree initially, it starts with the root of the tree, followed by the left and right nodes. If both the left and right child nodes are completed, then the left child node is turned to a folder and the file-name value is inserted within the folder of the left and right nodes. The same procedure continues for the right child node also. Finally, the leaf level contains the file names. The insertion and searching are different from the binary search tree. In binary search, insertion takes place by comparing the value in the root with the left and right. To search for a file in the proposed method, the index value of the file names is maintained globally in the controller node and locates exactly the level and the file where it is.

Insertion and searching

In a cloud storage system, a large amount of user files is stored on different compute nodes. When inserting a file in the cloud storage system, the compute node and controller node use a data structure algorithm. The algorithm is used to efficiently retrieve and insert the files in the storage volume. To insert and search a file, the first algorithm uses the digital bipartite algorithm. For every insertion of a file in the volume, the index will be stored in the controller node. Inserting a file for a particular user happens within the instance of a VM in the cloud. The initial request is stored under the root of the left and right nodes with the binary value of a file name with preceding 01, 10, and 11, which is converted into a decimal value and stored as a file name. The same procedure is repeated for all the next levels. The file name with a binary value and preceding bits are sent to the global index that is maintained in the controller node. The updates sent by the compute node are maintained in a controller node, ie, the global index list. While retrieving a file from the compute node, the request submitted by the user is sent to the controller, and the search takes place in the global index with the file name value and instance ID. If it matches with any of the files, then the request is sent to the corresponding compute node volume instead of searching all the compute node volumes.[12-15]

3.2 | Algorithm for insertion

```
    Get input (filename) from the user
                    Inserting function is invoked
    Case 1: If node equals NULL
                    Create a new node and insert it
    Case 2: If (node.left! = NULL && node.right! = NULL)
    Case 2a: if folder has two files
                                If (node.left.data.length > 3)&&node.right.data.length > 3)
                                        node = node.left
                                        node.left = new node (generate folder (root,node.left,data)
                                      node.left.left = tmp
                                        node.left.right = insert (node.left.right,data)
    Case 2b: if the folder has one folder and one file
                    Else if (node.left.data.length < 3)&&(node.right.data.length > 3)
                              node.tmp = node.right
                              node.right = new.node (generate folder (root,node.right.data)
                              node.right.left = tmp;
                              node.right.right = insert (node.right.right,data)
    Case 2c:if the folder has 2 folders
                    Elseif((node.left.data.length < 3) && (node.right.data.length < 3))
                    Else ((node.right. = insert (node.right,data)
    Case 3:if the current nodehas no folders
                    Elseif (node.left==null) then
                              node.left = insert (node.left.data);
                    Else node.right = insert (node.right,data)
            End insert
 Write the data into file serialization
 The file can be reconstructed as a tree using deserialization Print ancestor
```

Algorithm for Search

```
isExist (root,target)
{
        If (root==null) then
                Return false;
        If (root.left==target or root.right==target)
                Return true
Return false
}
```

Cloud infrastructure consists of many compute nodes. More number of cloud users stores their huge volume of files in the cloud storage. To store and retrieve the user files in a scalable and efficient manner, a good indexing method is required. Such indexing scheme should support parallel search to improve the efficiency. Here, the tree-based indexing method is used for organizing and managing the file storage. Storing and retrieving files in cloud storage is done by considering the key value. Trie is a better searching algorithm and is used in searching data from different sources such as big databases, data in compiler, and dictionaries, and in finding the router path in computer networks. Trie is a type of indexing method for searching and storing files efficiently. Compared with a binary search tree searching a file in a sorted order with a minimum number of searches, this is one of the methods to increase the quality of searching speed.

Client application for accessing the user storage is linked with the controller. The controller acts as an index server. The controller consists of compute node information, storage disk information, and instance information. The storage nodes are integrated with compute nodes. The storage index consists of a table where the key for the file name will be mapped to the instance name and compute nodes. When the user wishes to store the file in the accessed instance and the storage disk, the file name is first converted into a hash value. The hash value, accessed instance ID, and ID information of the accessed compute node are stored in the table. This index table is maintained in the controller node. This index table is responsible for processing the request submitted by the user to search and store a file in the cloud storage system.

To route the query submitted by the user among the compute nodes, all the compute nodes are connected to each other in a peer-to-peer manner.

3.3 | Digit compact prefix insertion and searching

First, consider the file insertion in the cloud storage disk, and through instance, the user can access the cloud while using the system; if the user wishes to store the file, the file name is converted into a unique hash value. The hash value is considered a key value and consists of a number from 0-9. Number from 0 to 9 is considered as the root node. Based on the key of the user, the path begins to traverse from the root node to a terminal or leaf node. The final leaf node represents the key value for a corresponding file, and every level of the path has label. The labels are marked with a digit. Trie is like a tree structure that has root nodes and element nodes. The Trie node starts with the root node containing an array of number with size 10, which has numbers from 0 to 9. When a file name is given by the user, a unique hash key value will be generated, and a corresponding hash value will be stored in the tree structure starting from the root node.[16-19]

For example, in the case of numeric keys, each node has an array of ten pointers to its branches, one for each of the ten numbers. The keys as file names are stored in the leaf nodes. The keys are generated from the hash table. To access a node containing a key, move down a sequence of branch nodes following the corresponding branch based on the numeric array composing the key. To access these nodes, track a path from the root depending on the numbers forming the key, until the appropriate node holding the key is reached. Thus, the depth of a node in a tree depends on the key.

The condition for inserting a file name in the storage consists of three conditions. First, if the key value for the file name is already present in the structure, the file searching begins from the root to the down level. Second, if the key value for the file name matches with only minimum digits, the files stored in each disk consists of more number of files names; the time taken to find a file in the disk is proportional to the number of files. If the file name is stored in the form of the text, the filename may be 10 to 15 characters or more. While spending time to search the text down every 10 or 15 levels to get the corresponding file name in this digit compact prefix, the hash value is generated for the given file name, and searching takes place only in the minimum levels compared with the existing searching.

Each compute node activates the algorithm whenever the user digit compact prefix store and retrieve the files in the cloud storage. To access the query, a local search is performed in the index table in the controller node. In the local search, the controller node finds in which compute nodes and instances the user file has been stored. The controller sends a request to the particular compute node to retrieve the file.

The amount of files searched in the tree structure is $|x|$ pointers to get to the node for the corresponding file. Each pointer takes $O(1)$ time to search a file. A trie with N nodes will require a space of $O(N.|X|)$ due to the pointers in each node.

Algorithm

```
Begin Insert (file name ID)
{
If file already exists it displays file exist
    Convert word to number array
    Begin loop
        Call subnode (ch)
            If (Subnode of root)
                then create folder
            Else
                Add that node in the tree
                Create the folder
    End loop
Create new file with the file name ID
}
Begin search (filename ID)
{
Convert file name into character array
    Call subnode (ch)
    Loop
    If it return null then return false
    Else
    Call subnode (ch)
    End
    If (current.is end==true)
    Then
    Return true
}
```

4 | IMPLEMENTATION AND RESULTS

The proposed work is implemented in OpenStack icehouse with four-node setup. The four-node setup consists of a controller and compute nodes. The controller node has an i7 processor, 16-GB memory, and a 1-TB hard disk, and the compute node has an i5 processor, 8-GB memory, and a 500-GB hard disk. The controller and compute nodes are shown in Figure 3. All the machines are connected through a 100-Mb/s Ethernet switch. The operating system in which OpenStack is installed is Ubuntu 14.0. Infrastructure as service was done to provide the storage to the user.

Cinder plays major role in OpenStack for block storage service. Block storage is used to provide storage resources to a client. The component for block storage is a user and a tenant (project), using role-based access assignment. User actions are controlled by the roles, and it is maintained and configured by the administrator. Each user's volume acts as secondary storage device similar to a hard disk and pen drives to store their files or data. The client can use their storage without any knowledge of where their storage is actually available. To provide the storage service, Cinder API, Cinder scheduler, and Cinder-volume services are configured in Cinder. These services are hosted in the controller, and volumes are attached in the compute node. The software-based Cinder plug-in is used for creating volume (ie, LVM) The number of instances, CPU, IPs, and volume hypervisor details are shown in Figure 3.

Before storing a file in the cloud server, a random unique number is generated for the given file name, and the file is loaded in the cloud server. The unique number of the file name, volume number, instance id, and compute node number are loaded in the global file in the controller node. The global file is used to reduce the search time. The time taken to store in the specified path and the time taken to retrieve in that particular path is calculated for bipartite and numeric trie algorithm and shown in a graph.

First, these two operations are compared with the existing binary search and trie algorithm with different file sizes. In the existing binary tree algorithm, if a file is inserted and there is no root, then the inserted file is stored in the root or it is placed on the left or right of the root. However, in the proposed algorithm, when a file is inserted, it is placed to the left and the right of the root. The leaf node contains the files. Inserting takes only less time. Searching in existing binary tree takes more time compared with the proposed bipartite algorithm. In the bipartite algorithm, when a file is given by the user, searching does not take place root by root; it directly finds the path of the file and retrieved. The existing algorithm can be a full binary tree and not a complete binary tree in all cases.

The existing trie algorithm stores the file by considering a file name letter by letter in each level. If the file name is of seven characters, the first letter is inserted in the root, the second letter in the second level, and so on. In the proposed method of the number trie algorithm, the file

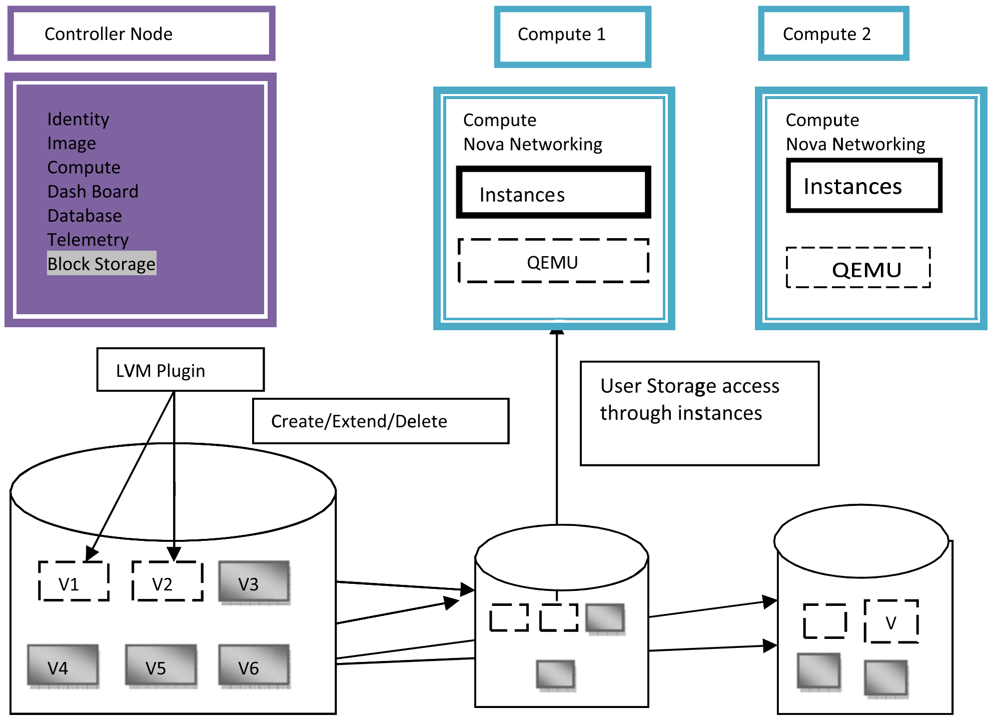

FIGURE 3 Components used in OpenStack

names equivalent of the hash value are considered. The same seven-character file name is converted into some 4 or 5 digits. The time taken to insert seven characters takes more time compared with inserting digits.

5 | PERFORMANCE EVALUATION

The performance evaluation of the proposed system is done by analyzing its fulfillment of file storage and its efficient retrieval. The digit compact prefix algorithm shows that they are indeed lightweight. We will focus on the overall throughput in terms of number of files processed, the time required to store the files in its logical structure, the time required to retrieve the file from storage, and the number of queries processed by our cloud experimental setup. The experiment is conducted using OpenStack (Icehouse release) on the Linux System with Intel Core i7 processor running at 3.24 GHz and 16-GB DDR3 RAM. Our multinode environment consists of one controller node and four compute nodes. We tabulate our result in Table 2 for compute node 1 and compute node 2, respectively. In addition to that, the user set (200 users) and the file set (250 files) represent a number of users and files processed vice versa. Figure 4 shows a number of files to be inserted using our first algorithm named digital bipartite. The result indicates that, for a given set of compute nodes, the number of files inserted in OpenStack storage increases exponentially within a minimum amount of time. This is because the number of compute nodes increases lead to processing multiple requests at a unit time. Similarly, the time to retrieve a file using digital bipartite is also a relatively minimal amount of time. So, we can retrieve a file within a given minimum unit of time. Figure 5 shows the effectiveness of the digital bipartite algorithm in terms of file retrieval in OpenStack multinode environment.

TABLE 2 Time comparison for two algorithms

Compute Node	Number of Users	Number of Files	File Size	Insertion Time		Searching Time		Number of Files Retrieved	
				Digital Bipartite	Digit Compact Prefix	Digital Bipartite	Digit Compact Prefix	Digital Bipartite	Digit Compact Prefix
Compute Node 1	50	50	1.1-2.5 Mb	2.142486	0.56	0.612663	0.01547	48	50
			3.5-5.5 Mb	2.424766	0.63	0.510123	0.018543		
			7-15 Mb	2.245755	0.63	0.594566	0.0169		
			20-93 Mb	2.557484	0.69	0.561066	0.009		
			95 Mb-1.6Gb	2.594475	0.69	0.559191	0.00836		
Compute Node 2	50	50	1.1-2.5 Mb	2.296403	0.54	0.468975	0.01422	49	50
			3.5-5.5 Mb	2.290065	0.55	0.568975	0.01849		
			7-15 Mb	2.300066	0.552	0.574898	0.01002		
			20-93 Mb	2.300645	0.552	0.433665	0.0159		
			95 Mb-1.6Gb	2.313312	0.545	0.486626	0.01		

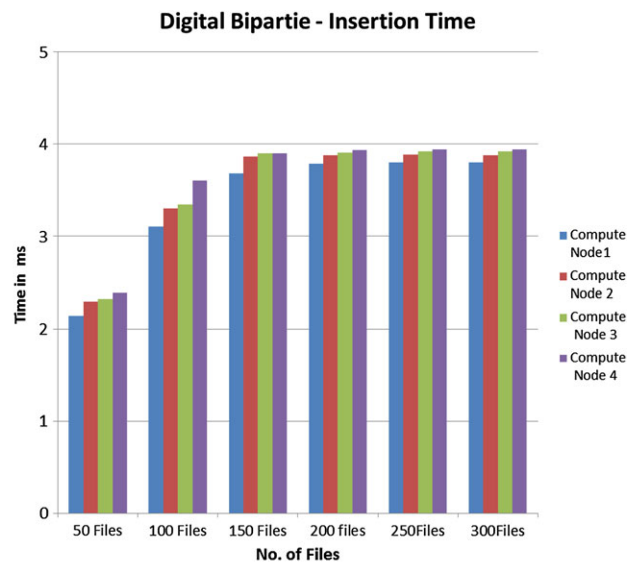

FIGURE 4 Insertion time for *N* files

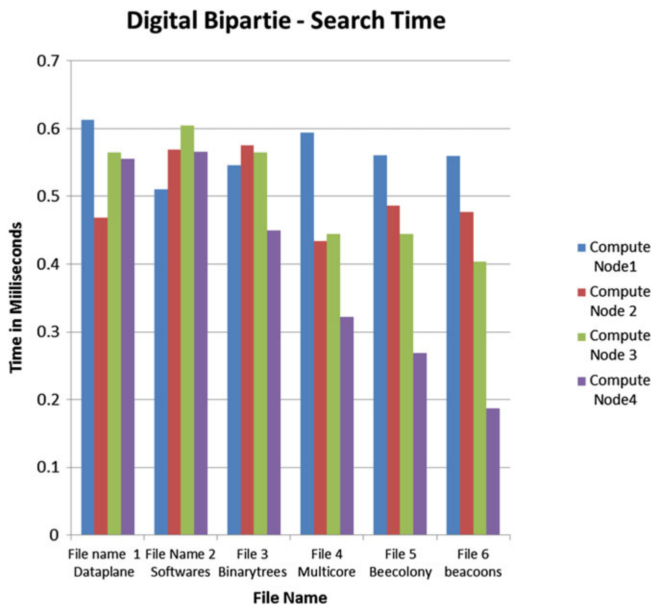

FIGURE 5 Search time for different file sizes

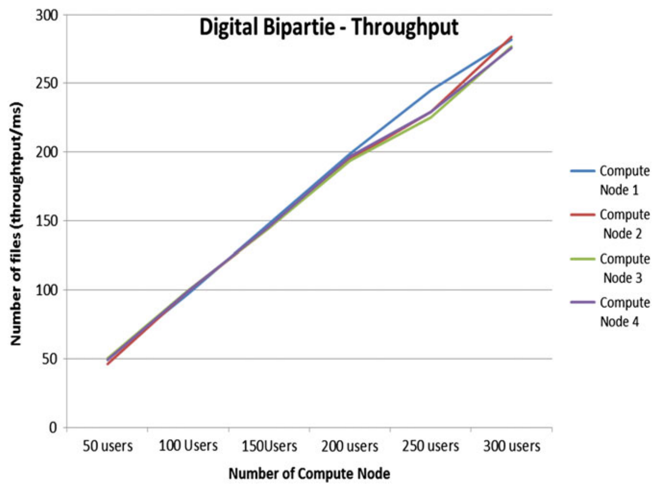

FIGURE 6 Throughput for *N* users

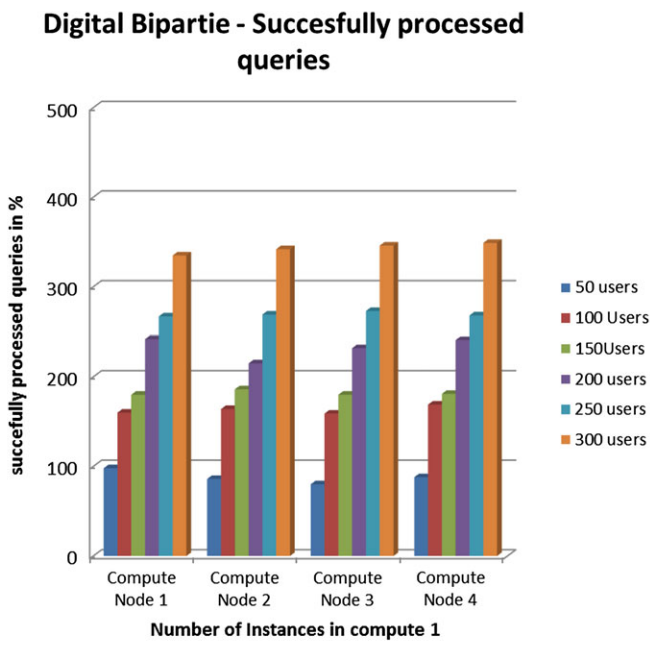

FIGURE 7 Successful queries processed by compute nodes

We then turn our focus to the throughput for certain users, ie, the total number of files processed for a given set of users. Specifically, it is a ratio of a number of files processed from a given set of queries to the number of users. Figure 6 estimates the relationship between a number of processed files and users in multimode environment. The stability of the system is analyzed by successfully processed queries while executing digital bipartite and digit compact prefix algorithms. The number of processed queries increases gradually with respect to users. Figure 7 indicates the increase using multiple computation nodes.

Our experimental result on digit compact prefix indicates immediate retrieval than our proposed digital bipartite–based retrieval. In our experiment, digit compact prefix–based insertion takes 1.58 ms for 250 files to insert the files in OpenStack storage, and file retrieval takes on average of 0.012 ms. The successfully processed queries, as well as overall throughput of the digit compact prefix algorithm is better than the digital bipartite.

Insertion time is calculated by the time the user submits the file name minus the time taken to place the file on the tree. In digital bipartite the time is calculated based on the following conditions:

1. If the folder has two files;
2. If the folder has one folder and one file;
3. If the folder has two folders;
4. If the current node has no folders.

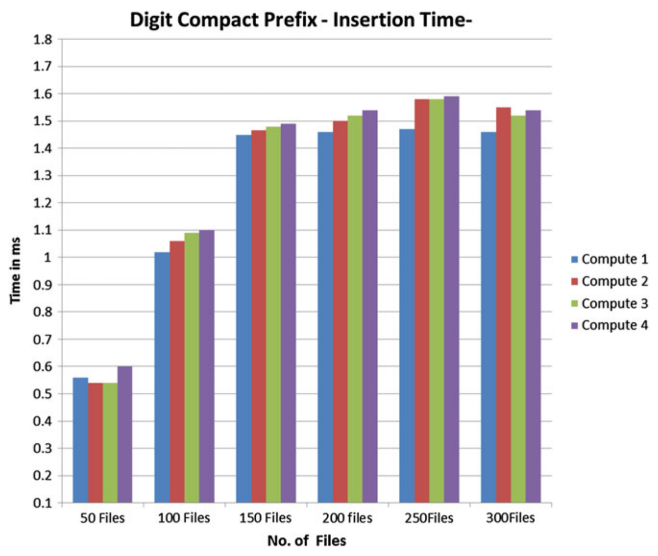

FIGURE 8 Insertion time for *N* files

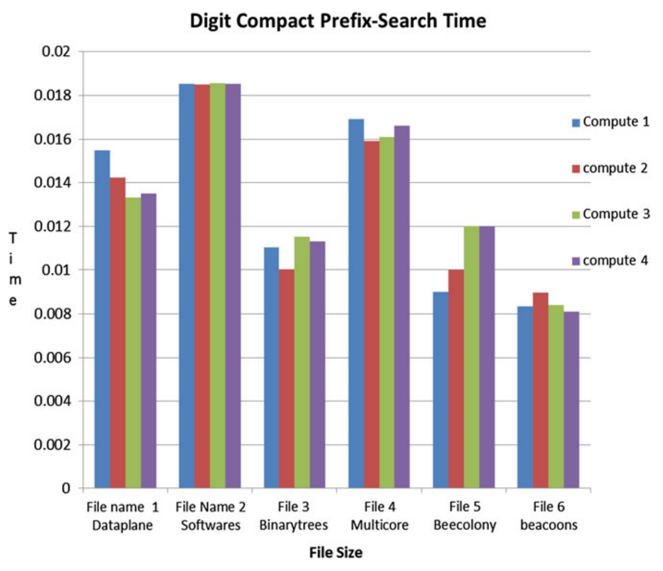

FIGURE 9 Search time for different file sizes

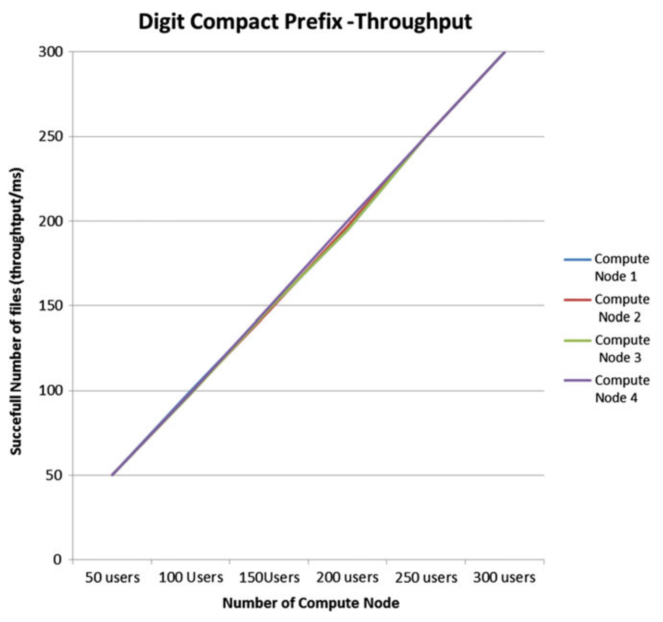

FIGURE 10 Throughput for *N* users

In digit compact prefix, the time is calculated based on the following conditions:

1. If the file name is not available
2. If the file name match with some words.

Search time is calculated by the time the user submits the file name minus the time taken to search in the index table.

The throughput of compute node is calculated by taking the number of queries submitted by the user in the controller node by the number of users accessing the node.

Figure 8-11 show the insertion time, search time, processed queries, and the throughput of the digit compact prefix algorithm. Figure 12 compares search time of the two algorithms, and it shows that the digit compact prefix takes less time in searching a file. Figure 13 compares the insertion time of the two algorithms, and it shows that the digit compact prefix takes less time in inserting a file since the already available letters in a file name are not necessary to create a node. Table 2 shows that the average number of requests submitted by 50 users in compute nodes 1 and 2 for retrieving and inserting a file ranges from 1.1 MB to 2 GB. The time taken for each compute node is also shown in the table.

As expected, the running time of computation for both the algorithms increases logarithmically with a maximum number of files inserted in our system. Note that our experiment dealt with a number of compute nodes up to four in the OpenStack setup.

Finally, we remark that, based on our experimental outcomes, the proposed two algorithms, namely digital bipartite and digit compact prefix, achieve efficient file storage and retrieval with maximum throughput.

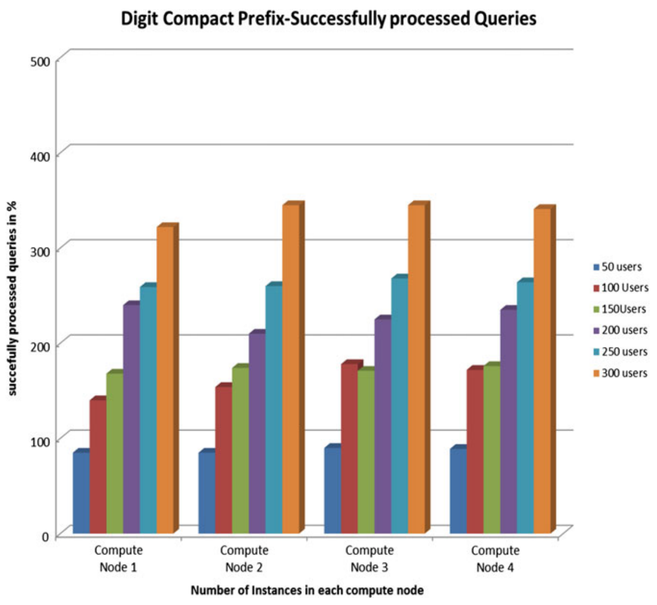

FIGURE 11 Successful queries processed by compute nodes

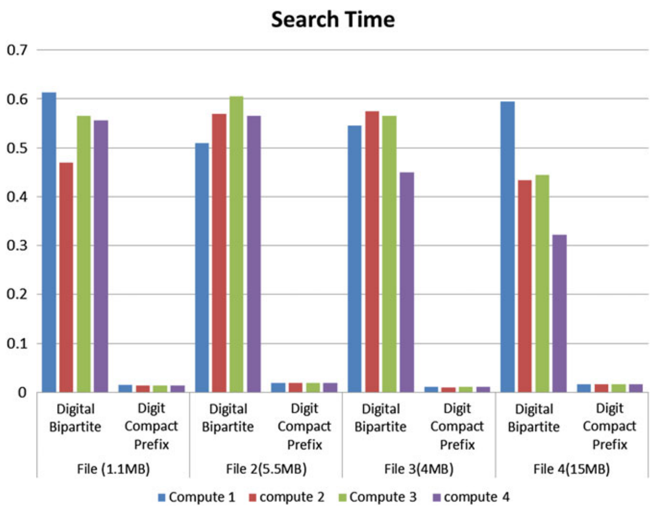

FIGURE 12 Comparing search time: two algorithms

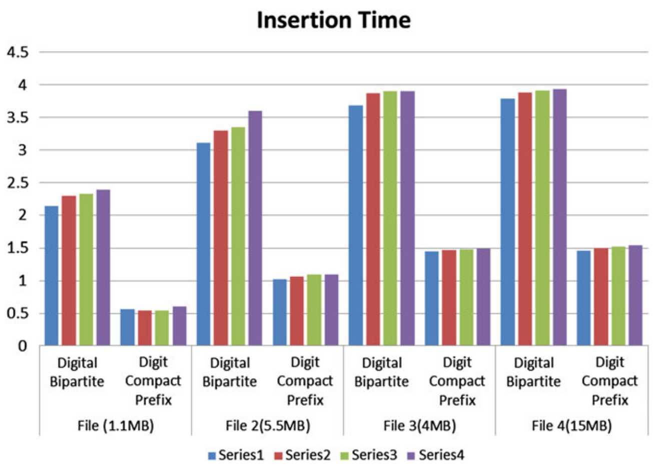

FIGURE 13 Comparing insertion time: two algorithms

6 | CONCLUSION

We have presented the implementation of an indexing technique for storing and retrieving files for infrastructure-as-a-service applications in cloud environment. We started by revisiting the traditional file system in application workloads and technological situations. Our implementation optimizes for vast files that are inserted to and then read in the file system. Our system provides a large amount of data transfer to many clients while reading and writing using the digit compact prefix algorithm, in comparison with digital bipartite algorithm. The work of the controller is minimized by creating an index. This proposed method has been implemented and evaluated in OpenStack, which is a real-world cloud infrastructure.

ORCID

P. Priya Ponnuswamy https://orcid.org/0000-0002-1690-1503

REFERENCES

1. He J, Wu Y, Fu Y, Zhou W. Clone-based data index in cloud storage system. Paper presented at: 4th International Conference on Mechatronics, Manufacturing and Materials Engineering (MME 2016); 2016; Wuhan, China.
2. Li D. A novel distributed index method for cloud computing. *Int J Grid Distrib Comput*. 2016;9(2):1-16.
3. Lu Z, Shen G. Towards a DHT networks and safe file sharing scheme in cloud storage service. *Adv Sci Tech*. 2015;95:188-194.
4. Zhou W, Liu L, Jing H, Zhang C, Yao S, Wang S. K-gram based fuzzy keyword search over encrypted cloud computing. *J Softw Eng Appl*. 2013;6(1):29-32.
5. Wang C, Cao N, Ren K, Lou W. Enabling secure and efficient ranked keyword search over outsourced cloud data. *IEEE Trans Parallel Distrib Syst*. 2012;23(8):1467-1479.
6. Pagh R, Rodler FF. Cuckoo hashing. *J Algorithms*. 2004;51(2):122-144.
7. Wang J, Wu S, Gao H, Li J, Ooi BC. Indexing multi-dimensional data in a cloud system. Paper presented at: 2010 ACM SIGMOD International Conference on Management Data; 2010; Indianapolis, IN.
8. Zhang Q, Pan X, Shen Y, Li W. A novel scalable architecture of cloud storage system for small files based on P2P. Paper presented at: 2012 IEEE International Conference on Cluster Computing Workshops; 2012; Beijing, China.
9. Wu S, Jiang D, Ooi BC, Wu K-L. Efficient B-tree based indexing for cloud data processing. *Proceedings VLDB Endowment*. 2010;3(1-2):1207-1218.
10. Shirmarz A, Sabaei M, Hosseini M. Evaluation and comparison of binary trie base IP lookup algorithms with real edge router IP prefix dataset. *Int J Adv Comput Sci Appl*. 2016;7(6):155-161.
11. Ghemawat S, Gobioff H, Leung S-T. The Google file system. Paper presented at: 9th ACM Symposium on Operating Systems Principles; 2003; Bolton Landing, NY.
12. He J, Wu Y, Dong Y, Zhang Y, Zhou W. Dynamic multidimensional index for large-scale cloud data. *J Cloud Comput Adv Syst Appl*. 2016;5:10.
13. Plavec F, Vranesic ZG, Brown SD. On digital search tree: simple method for constructing balanced binary tree. Paper presented at: 2nd International Conference on Software and Data Technologies; 2007; Barcelona, Spain.
14. Koutrika G, Zadeh ZM, Garcia-Montina H. Data Clouds: summarizing keyword search results over structured data. Paper presented at: 12th International Conference on Extending Database Technology: Advances in Database Technology; 2009; Saint Petersburg, Russia.
15. Bagwell P. Fast and space efficient trie searches. https://infoscience.epfl.ch/record/64394/files/triesearches.pdf. Published 2000.
16. Zhelev R, Georgier V. A DHT based scalable and fault tolerant cloud information service. Paper presented at: The 5th International Conference on Ubiquitous Computing, Systems, Services, and Technologies; 2011; Lisbon, Portugal.
17. Nagaraj K, Khandelwal H, Killian C, Rao Kompella R. Hierarchy aware distributed overlays in data centers using DC2. Paper presented at: 4th International Conference on Communication Systems and Networks (COMSNETS 2012); 2012; Bangalore, India.
18. Wu S, Wu K-L. An indexing framework for efficient retrieval on the cloud. *IEEE Data Eng Bull*. 2009;32(1):75-82.
19. Shekokar N, Sampat K, Chanwalla C, Shah J. Implementation of fuzzy keyword search over encrypted data in cloud computing. *Procedia Comput Sci*. 2015;45:499-505.

How to cite this article: Priya Ponnuswamy P, Vidhya Priya R, Shabari Ram CP. File retrieval and storage in the open source cloud tool using digital bipartite and digit compact prefix indexing method. *Concurrency Computat Pract Exper*. 2019;e5307. https://doi.org/10.1002/cpe.5307